Bulla Factoring & Core Contracts V2 Audit Report

Prepared by LPHAC

Version 1.1

Lead Auditors

0xleastwood

0xkamensec

September 4, 2025

Contents

1 About LPHAC

LPHAC is a Web3 security organization aiming to protect projects and their partners against malicious actors. We aim to provide holistic improvements to a project's security stack in a safe and realiable way.

2 Disclaimer

The LPHAC team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Bulla Network enables the on-chain creation of credit pools for invoice factoring. By adhering to the **ERC4626** specification, permissioned depositors can earn interest for facilitating invoice funding. Through these contracts, invoice issuers can factor their receivables, allowing them to receive early payments in exchange for a premium. This integration not only broadens the utility of the Bulla Claim Protocol but also provides a new financial mechanism for liquidity and credit management on-chain.

This audit update introduces *several* new features:

- The ability to queue redemptions/withdrawals from the pool in a FIFO order.
- A new controller pattern which allows for specialized contracts to completely control the claims they create, enabling:
  - Extended functionality: beyond basic claim operations (interest calculations, complex payment flows, etc.)
  - Custom business logic: for specific use cases (lending, invoicing, etc)
  - Additional states and workflows: tailored to different financial instruments
  - Domain-specific features: while maintaining core claim properties
- The protocol uses an EIP712-based approval system that allows users to grant specific permissions to controllers without requiring multiple transactions. This enables gasless interactions and streamlined user experiences while maintaining security.

These new features make Bulla Network much more extensible as the team can introduce new contracts which inherit the base controller implementation.

5 Audit Scope

The audit was started on commits 063ed28 and 256561d for **7** days. A subsequent fix review was conducted on final commits b003dca and 070653f that went for **1** day.

The following contracts were included as part of the review scope, including any related external dependencies:

5.1 factoring-contracts

```
factoring-contracts/
|-- contracts/
    |-- BullaClaimV2InvoiceProviderAdapterV2.sol
    |-- BullaFactoring.sol
    |-- RedemptionQueue.sol
    \-- interfaces/
        |-- IBullaFactoring.sol
        |-- IERC20.sol
        |-- IInvoiceProviderAdapter.sol
        \-- IRedemptionQueue.sol
```

5.2 bulla-contracts-v2

```
bulla-contracts-v2/
|-- src/
    |-- BullaApprovalRegistry.sol
    |-- BullaClaimControllerBase.sol
    |-- BullaClaimV2.sol
    |-- BullaControllerRegistry.sol
    |-- BullaFrendLendV2.sol
    |-- BullaInvoice.sol
    |-- interfaces/
    |   |-- IBullaApprovalRegistry.sol
    |   |-- IBullaClaimAdmin.sol
    |   |-- IBullaClaimCore.sol
    |   |-- IBullaClaimV2.sol
    |   |-- IBullaControllerRegistry.sol
    |   |-- IBullaFrendLendV2.sol
    |   |-- IBullaInvoice.sol
    |-- libraries/
    |   |-- BullaClaimPermitLib.sol
    |   |-- BullaClaimValidationLib.sol
    |   \-- CompoundInterestLib.sol
    \-- types/
        \-- Types.sol
```

6 Executive Summary

Over the course of **7** days, the LPHAC team conducted an audit on the Bulla Factoring & Core Contracts V2 smart contracts provided by Bulla Network. In this period, a total of **21** issues were found.

Summary

Project Name	Bulla Factoring & Core Contracts V2
Repository 1	factoring-contracts
Initial Commit	063ed28aae97...
Final Commit	b003dca99a90...
Repository 2	bulla-contracts-V2
Initial Commit	256561d6a7a2...
Final Commit	070653fe0e54...
Audit Timeline	Aug 11th - Aug 19th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	7
Medium Risk	4
Low Risk	6
Informational	3
Gas Optimizations	1
Total Issues	21

Summary of Findings

[H-1] Protocol fees can be DoS'd by adding arbitrary tokens to `protocolFeeTokens`	✓ Verified Fix
[H-2] Redemption queue can be DoS'd through an "out-of-gas" error	✓ Verified Fix
[H-3] Queue processing may remove a redemption at the wrong index leading to deadlock	✓ Verified Fix
[H-4] Invoice impairments do not consider already paid amounts impacting the capital account balance	✓ Verified Fix
[H-5] Core protocol fees can be stolen due to missing `payClaim` validation	✓ Verified Fix
[H-6] `BoringBatchable` library usage allows for `msg.value` re-use attacks	✓ Verified Fix
[H-7] Protocol fee balances can be corrupted by abusing callback logic	✓ Verified Fix
[M-1] `_markClaimAsPaid` fails to transfer the NFT to the debtor when the claim is paid	✓ Verified Fix
[M-2] Creditors using native token can deny loan repayments increasing `accruedInterest`	✓ Wontfix
[M-3] Inconsistent elapsed days calculation when computing simple interest causes accounting issues	✓ Verified Fix

[M-4] Loan acceptance fees can be bypassed by abusing batch logic	✓ Verified Fix
[L-1] Incorrect grace bound check on claim impairment	✓ Verified Fix
[L-2] Creating claims and loans skips the first `ID`	✓ Verified Fix
[L-3] `getClaim` results in creditor and debtor inconsistencies after settlement	✓ Verified Fix
[L-4] Loans offered via `BullaFrendLendV2` cannot be unfactored	✓ Wontfix
[L-5] `gracePeriodDays` cannot be changed while there are active invoices	✓ Verified Fix
[L-6] Invoice unfactoring does not reconcile all paid invoices before processing redemption queue	✓ Verified Fix
[I-1] `dueBy` parameter in `CompoundInterestLib` should be renamed	✓ Verified Fix
[I-2] Redundant `numberOfPeriodsPerYear` check in `offerLoan`	✓ Verified Fix
[I-3] Redundant transfer made in `_acceptLoan`	✓ Verified Fix
[G-1] Arrays are redundantly iterated through twice	✓ Verified Fix

7 Findings

7.1 High

7.1.1 Protocol fees can be DoS'd by adding arbitrary tokens to `protocolFeeTokens`

Context: BullaFrendLendV2.sol#L426-L440, BullaFrendLendV2.sol#L380, BullaInvoice.sol#L493-L514, BullaInvoice.sol#335

Description: Fee withdrawal in `BullaFrendLendV2` and `BullaInvoice` uses `for` loops to iterate through `protocolFeeTokens` and send all collected fees to the admin account. However, any arbitrary token can be used as a `claim.token`, which is ultimately passed as a valid `protocolFeeToken`. Malicious tokens risk token withdrawal failing completely.

During loan repayment fee tokens are added as follows:

```
function queueRedemption(
    address owner,
    address receiver,
    uint256 shares,
    uint256 assets
) external onlyFactoringContract returns (uint256 queueIndex) {
    if (owner == address(0)) revert InvalidOwner();
    if (receiver == address(0)) revert InvalidReceiver();
    if ((shares > 0) == (assets > 0)) revert InvalidRedemptionType();

    QueuedRedemption memory redemption = QueuedRedemption({
        owner: owner,
        receiver: receiver,
        shares: shares,
        assets: assets
    });

    queueIndex = queue.length;
    queue.push(redemption);

    emit RedemptionQueued(owner, receiver, shares, assets, queueIndex);

    return queueIndex;
}
```

By allowing any arbitrary token, attackers can control execution flow of fee withdrawal, choosing to revert all calls to `withdrawAllFees()` in order to DoS fee retrieval.

Recommendation: Use token whitelisting on all token interactions, additionally allow for individual fee token claims.

Bulla: Resolved in PR 90 and PR 98.

LPHAC: Verified fixes. The protocol fee is accumulated for all tokens but `withdrawAllFees()` now only transfers out owner whitelisted tokens, avoiding any potential DoS vector.

7.1.2 Redemption queue can be DoS'd through an "out-of-gas" error

Context: BullaFactoring.sol#L1104-L1233

Description: Recent changes have been made to the `BullaFactoring` contract which allow for share/asset redemptions/withdrawals to be executed through a FIFO queue which will process queued redemptions when liquidity becomes available.

The `_processRedemptionQueue()` function implements the processing logic which will iterate through queued redemptions and process withdrawals until the pool no longer has available liquidity or the queue is empty. There are *two* paths where a depositor could intentionally or unintentionally DoS the pool and prevent almost all user actions. These being:

- A depositor makes a large number of withdrawal/redemption requests with negligible amounts which will all be queued. When `_processRedemptionQueue()` attempts to process these, it will run out of gas.
- Alternatively, the above approach can be replicated by making a withdrawal/redemption request and moving shares to another account, causing the prior request to be cancelled.
- A larger number of redemptions requests are pushed onto the queue and subsequently cancelled, leaving no-op redemptions which the queue head must iterate through before executing other user requests.

```
function queueRedemption(
    address owner,
    address receiver,
    uint256 shares,
    uint256 assets
) external onlyFactoringContract returns (uint256 queueIndex) {
    if (owner == address(0)) revert InvalidOwner();
    if (receiver == address(0)) revert InvalidReceiver();
    if ((shares > 0) == (assets > 0)) revert InvalidRedemptionType();

    QueuedRedemption memory redemption = QueuedRedemption({
        owner: owner,
        receiver: receiver,
        shares: shares,
        assets: assets
    });

    queueIndex = queue.length;
    queue.push(redemption);

    emit RedemptionQueued(owner, receiver, shares, assets, queueIndex);

    return queueIndex;
}
```

While it's important to note that `setRedemptionQueue()` is able to configure a new `redemptionQueue` contract, it may prove incredibly difficult to remedy the bloated queue state.

Recommendation: Considering pool depositors will be limited to a small set of users, allowing only one (or two entries if we need to support separate share/asset redemptions) queue entry per user should prevent any abuse of the current implemented redemption queue logic. If a user attempts to increase the number of shares or assets in their request, then the user's redemption will be moved to the end of the queue as if it was cancelled and made anew.

Bulla: Resolved in PR 141.

LPHAC: I think this is still problematic because we do not compact the queue and due to FIFO design, if we cancel all existing queue redemptions, they are zeroed out but not necessarily removed because the head only advances if a cancelled queue item is at the head. Meaning it's still possible to DoS by being the 2nd person in a queue and queueing a withdrawal/redemption, cancelling and then repeating this again and again.

Bulla: Further resolved in PR 145.

LPHAC: Verified fixes. On new redemption requests the queue is compacted to remove any previously cancelled items. Because the set of depositors is small, this design is within block gas limits and ensures each depositor only has one active queue item.

7.1.3 Queue processing may remove a redemption at the wrong index leading to deadlock

Context: BullaFactoring.sol#L1162-L1233, RedemptionQueue.sol#L76-L97

Description: When an owner has insufficient funds to satisfy their redemption request in `_processRedemptionQueue()`, the owner's request is subsequently cancelled and removed from queue. The head is advanced to the next valid item if the request's queue index was equal to the queue head.

However, there is a severe mistake made here where `redemptionQueue.cancelQueuedRedemption(0)` will always attempt to cancel the queued redemption at index zero which may or may not be the queue head. Consequently, the queue head may never be advanced and we reach a deadlock where the `while` loop condition in `_processRedemptionQueue()` never terminates.

Recommendation: The `_processRedemptionQueue()` function needs to be updated to properly remove the current redemption request. This could be by calling `removeAmountFromFirstOwner()` without executing the withdrawal.

Bulla: Resolved in PR 140.

LPHAC: Verified fix. The owner's redemption request is removed from the queue by processing a no-op redemption/withdrawal which removes the item from the queue without impacting the user's shares/assets.

7.1.4 Invoice impairments do not consider already paid amounts impacting the capital account balance

Context: BullaFactoring.sol#L394-L432

Description: An invoice can be impaired when the current `block.timestamp` has exceeded `invoice.dueDate + (gracePeriodDays * 1 days)` and must be recording by calling `impairInvoice()` to have the invoice removed from the `activeInvoices` list and added to the `impairedByFundInvoicesIds` list.

The `calculateRealizedGainLoss()` function is used to calculate the capital account balance so effectively when an invoice is impaired, the loss is realized by all depositors equally. However, this fails to track any payments made since funding started. This can be tracked through the `getPaymentsOnInvoiceSinceFunding()` function to have this included as part of the `realizedGains` in `calculateRealizedGainLoss()`.

```
function calculateRealizedGainLoss() public view returns (int256) {
    int256 realizedGains = 0;
    // Consider gains from paid invoices
    for (uint256 i = 0; i < paidInvoicesIds.length; i++) {
        uint256 invoiceId = paidInvoicesIds[i];
        realizedGains += int256(paidInvoicesGain[invoiceId]);
    }

    // Consider losses from impaired invoices by fund
    for (uint256 i = 0; i < impairedByFundInvoicesIds.length; i++) {
        uint256 invoiceId = impairedByFundInvoicesIds[i];
        realizedGains += int256(impairments[invoiceId].gainAmount);
        realizedGains -= int256(impairments[invoiceId].lossAmount);
    }

    // Consider losses from impaired invoices in activeInvoices
    for (uint256 i = 0; i < activeInvoices.length; i++) {
        uint256 invoiceId = activeInvoices[i];
        if (isInvoiceImpaired(invoiceId)) {
            uint256 fundedAmount = approvedInvoices[invoiceId].fundedAmountNet;
            realizedGains -= int256(fundedAmount);
        }
    }
```

```
    return realizedGains;
}
```

Recommendation: Consider modifying `impairInvoice()` to call `getPaymentsOnInvoiceSinceFunding()` and retrieve the up-to-date paid amount on the invoice. It's entirely possible that the total interest and principal amount paid is greater than `fundedAmountNet` and hence we might not need to impair the invoice but could instead unfactor the invoice. With the correct `gainAmount` and `lossAmount` recorded, `calculateCapitalAccount()` should track the correct balance.

Bulla: Resolved in PR 144.

LPHAC: Verified fix. `calculateRealizedGainLoss()` has been modified to consider any paid amount made on the invoice before it was impaired.

7.1.5 Core protocol fees can be stolen due to missing `payClaim` validation

Context: BullaClaimV2.sol#L224-L274, BullaClaimV2.sol#L620-L627

Description: The `BullaClaimV2` contract facilitates claim creations, status management and payments to creditors. There is a single `payClaimFromControllerWithoutTransfer()` function which explicitly allows for the claim state to be updated without any transfer to the creditor. This is only possible when called from `claim.controller` and this is always properly set from the perspective of the `BullaFrendLendV2` and `BullaInvoice` contracts.

When an invoice has been created with no controller, payments must be made via `payClaim()` which sends native ether to the creditor when `claim.token == address(0)` or otherwise performs `ERC20(claim.token).safeTransferFrom()`. The latter is trusted because `from == msg.sender` but the native token transfer must verify/validate that `msg.value == paymentAmount` which it fails to do at any point in time in the execution flow. Consequently, a malicious claim creator could steal accumulated core protocol fees.

```
function withdrawAllFees() external onlyOwner {
    uint256 ethBalance = address(this).balance;
    if (ethBalance > 0) {
        (bool success,) = payable(owner()).call{value: ethBalance}("");
        if (!success) revert WithdrawalFailed();
        emit FeeWithdrawn(owner(), ethBalance);
    }
}
```

The setup for this would be to create a malicious claim where the claim's payment asset is set to use native ether and we are both the debtor and creditor. Then we are able to call `payClaim()` and receive the `paymentAmount` as native ether via the `creditor.safeTransferETH(paymentAmount)` call in `_payClaim()`.

Recommendation: Consider updating `_payClaim()` to validate `msg.value` when `claim.token == address(0)`.

Bulla: Resolved in PR 96.

LPHAC: Verified fix. `msg.value` is now being validated when `claim.token == address(0)` as per the suggested recommendation.

7.1.6 `BoringBatchable` library usage allows for `msg.value` re-use attacks

Context: BullaFrendLendV2.sol#L259-L262, BullaInvoice.sol#L267-L352

Description: The inheritance of the `BoringBatchable` library introduces a known issue with any check using `msg.value`. When performing a `address(this).delegatecall(calls[i])` call and the target function uses `msg.value` in any meaningful way, we can perform a set of batch calls where `msg.value` is re-used multiple times across several calls.

There are *three* key instances where this issue can be abused:

1. To perform a fee bypass in `_createInvoice()`, `batchCreateInvoices()` or `_acceptLoan()`.
    - `_createClaim()`
2. To abuse `payInvoice()` and have native ether transferred to a malicious creditor out of accumulated protocol fees.
3. To create claims and only pay the protocol fee once.

The second instance is the most concerning because the malicious invoice creator is effectively able to steal accumulated protocol fees.

Proof of Concept: The `testAcceptLoansViaBatch()` demonstrates a fee bypass vulnerability in the `BullaFrendLendV2` contract.

When using the generic `batch()` function with individual `acceptLoan()` calls:

- Each `acceptLoan()` call receives the same `msg.value` due to the delegate call behaviour.
- Both loans get accepted but only a single fee (0.01 ETH) is paid instead of the required two fees (0.02 ETH).
- The debtor receives 3 ETH worth of loans (1 ETH + 2 ETH) while only paying the fee once.

```
// This test demonstrates a potential vulnerability where batched delegate calls to acceptLoan()
// reuse msg.value between each call, potentially allowing fee bypass if the contract had sufficient
↪  balance
function testAcceptLoansViaBatch() public {
    // Setup: Creditor creates multiple loan offers
    vm.startPrank(creditor);
    weth.approve(address(bullaFrendLend), 3 ether);

    uint256 loanId1 = bullaFrendLend.offerLoan(
        new
↪   LoanRequestParamsBuilder().withCreditor(creditor).withDebtor(debtor).withToken(address(weth))
            .withLoanAmount(1 ether).build()
    );
    uint256 loanId2 = bullaFrendLend.offerLoan(
        new
↪   LoanRequestParamsBuilder().withCreditor(creditor).withDebtor(debtor).withToken(address(weth))
            .withLoanAmount(2 ether).build()
    );
    vm.stopPrank();

    // Setup permissions for loan acceptance
    _permitAcceptLoan(debtorPK, 2);

    // Add ETH balance to the contract to avoid OutOfFunds error and demonstrate the vulnerability
    vm.deal(address(bullaFrendLend), 1 ether);

    // Record initial WETH balance
    uint256 debtorInitialBalance = weth.balanceOf(debtor);

    bytes[] memory calls = new bytes[](2);
    calls[0] = abi.encodeCall(BullaFrendLendV2.acceptLoan, (loanId1));
    calls[1] = abi.encodeCall(BullaFrendLendV2.acceptLoan, (loanId2));
```

```
    // Debtor accepts both loans with only a single fee attached
    vm.prank(debtor);
    bullaFrendLend.batch{ value: FEE }(calls, true);

    // Verify that BOTH loans were accepted with only ONE fee payment (vulnerability demonstrated)
    assertEq(bullaClaim.currentClaimId(), 2, "Both loans should have been accepted");

    Claim memory claim1 = bullaClaim.getClaim(1);
    Claim memory claim2 = bullaClaim.getClaim(2);
    assertEq(claim1.debtor, debtor, "First loan should have been accepted");
    assertEq(claim2.debtor, debtor, "Second loan should have been accepted");

    // Verify debtor received tokens from both loans
    assertEq(weth.balanceOf(debtor), debtorInitialBalance + 3 ether, "Debtor should have received both
↪   loan amounts");
}
```

Recommendation: This only scratches the surface of this issue as there are other much more severe examples that would need to be explored but ultimately this library should not be used whenever a contract is using `msg.value` in any meaningful way.

Bulla: Resolved in PR 92.

LPHAC: Verified fix. The `BoringBatchable` library has been removed from all affected contracts.

7.1.7 Protocol fee balances can be corrupted by abusing callback logic

Context: BullaFrendLendV2.sol#L328-L331, BullaFrendLendV2.sol#L572-L581

Description: The `BullaFrendLendV2` contract allows for an loan offer creator to designate a target callback address and function selector. This callback is executed only when the loan offer has been accepted by a counterparty where `_executeCallback()` will be called.

Because loans can be offered and accepted by any user, malicious actors could corrupt `protocolFeeTokens` balances by performing a callback to `token.transfer(address to, uint256 value)` where the `address to` and `uint256 value` parameters are decoded from `loanOfferId` and `claimId` values respectively. Even though these values are minuscule, the impact is quite severe because `withdrawAllFees()` will not longer be callable it expects the fee amount store in `protocolFeesByToken[token]` to be readily available.

Note: This can be remedied by transferring additional tokens into the contract and then withdrawing but this is not an ideal method.

Recommendation: Do not allow for arbitrary callbacks to be executed. Callback targets should be whitelisted or blacklisted to prevent sensitive calls from being made.

Bulla: Resolved in PR 94 and PR 97.

LPHAC: Verified fixes. Callback targets and function selectors are now whitelisted by the contract owner. Unnecessary validation of this has been removed in favour of a single whitelist check when a target callback address is configured.

7.2 Medium

7.2.1 `_markClaimAsPaid` fails to transfer the NFT to the debtor when the claim is paid

Context: BullaClaimV2.sol#L439-L453

Description: When creating a claim on Bulla Network the claim can specify whether to transfer ownership of the claim to the debtor when the claim is finalized. This is managed through the `claim.payerReceivesClaimOnPayment`, however, when a payment is marked as paid rather than paid directly, the transfer never occurs.

Recommendation: Opt for pull over push patterns, where the user is able to request the NFT upon successful payment completion. This will consolidate the behaviour to a single point in the code, which will simplify verification of access control and conditions needed to execute the transfer.

Bulla: Resolved in PR 88.

LPHAC: Verified fix. The claim NFT is no longer transferred to the creditor so this inconsistency no longer exists.

7.2.2 Creditors using native token can deny loan repayments increasing `accruedInterest`

Context: BullaInvoice.sol#L328

Description: The `BullaInvoice` contract allows for the use of native tokens to be used during settlement. Native token usage increases attack surface area since different recipients may respond differently, and unexpectedly during receipt of those tokens.

During settlement if a smart contract is used as the `claim.creditor`, that a smart contract can control the success of attempts made to transfer eth to its address. Since the `payInvoice()` function, a critical function used to settle outstanding debts, calls the creditor as follows:

```
// Handle ETH payments
// Protocol fee for ETH stays in contract for admin withdrawal
if (creditorTotal > 0) {
    creditor.safeTransferETH(creditorTotal);
}
```

It is possible that the SC may intentionally revert here, forcing all invoice repayments to revert. By not allowing repayments to succeed a malicious creditor may increase `accruedInterest` that needs to be paid off.

Recommendation: All native token transfers should be forcefully wrapped with a token such as WETH, this will ensure call execution abides by expectations with limited edge cases.

Bulla: I'm not convinced we should fix this. We wouldn't want to pay a malicious creditor anyways, invoice can simply be rejected and move on to something else.

LPHAC: Noted and marked as acknowledged. Any accidental configuration will be documented.

7.2.3 Inconsistent elapsed days calculation when computing simple interest causes accounting issues

Context: CompoundInterestLib.sol#L125-L137, BullaFactoring.sol#L384

Description: The `BullaFactoring` contract has been modified to to accommodate loan offers via the `BullaFrendLendV2` contract. Regardless if the loan offered through this contract is calculating simple interest or compounding it, the `calculateFees()` function should expect `daysOfInterest` to never be more than the days calculated in the `CompoundInterestLib`. Otherwise, this may corrupt accounting because fees are overstated when the invoice is paid and reconciled.

```
uint256 daysSinceFunded = (block.timestamp > approval.fundedTimestamp) ? Math.mulDiv(block.timestamp -
↪  approval.fundedTimestamp, 1, 1 days, Math.Rounding.Ceil) : 0;
```

As per the above implementation in `BullaFactoring`, `daysSinceFunded` will round up to the nearest day and then call `calculateFees()` based on the result whereas `_computeSimpleInterest()` will truncate down.

```
// Calculate complete days elapsed (interest only accrues on complete days)
uint256 daysElapsed = secondsElapsed / 1 days;
```

Recommendation: Either the `BullaFactoring` or `CompoundInterestLib` implementation needs to be changed to match the other and maintain consistency throughout.

Bulla: Resolved in PR 142 and PR 143.

LPHAC: Verified fixes. `BullaFactoring` interest day calculations now round down to match `CompoundInterestLib`.

7.2.4 Loan acceptance fees can be bypassed by abusing batch logic

Context: BullaFrendLendV2.sol#L259-L331, BullaFrendLendV2.sol#L456-L522

Description: The `BullaLendFrendV2` contract makes use of a `_inBatchOperation` variable to skip fee validation when in a batch operation because `_validateBatch()` will verify that `msg.value == totalRequiredFee`.

However, by abusing re-entrancy in the `_acceptLoan()` function, a user could accept multiple loan offers and only pay the fee once. There are *two* ways where `_acceptLoan()` gives the caller control over the flow of execution:

1. The designated callback executed towards the end of the call.
2. Through an arbitrary `offer.params.token` which has custom logic on `safeTransferFrom()` or `safeTransfer()`.

So the approach of attack would be to call `batchAcceptLoans()`on one malicious loan created by the exploiter, then via the above re-entrant paths the user could make their own batch call to accept an arbitrary number of loans and skip any fee validation in `_acceptLoan()`.

Recommendation: Either whitelist tokens and callback targets or mark affected functions with the `nonReentrant` modifier.

Bulla: Resolved in PR 92.

LPHAC: Verified fix. The affected custom batch functions have now been removed altogether.

7.3 Low

7.3.1 Incorrect grace bound check on claim impairment

Context: BullaClaimValidationLib.sol#L143

Description: Incorrect grace period bound check. The check will not revert with `StillInGracePeriod` when `block.timestamp == claim.impairmentGracePeriod`, which is still technically within the grace period. The problematic condition is:

```
block.timestamp < claim.dueBy + claim.impairmentGracePeriod
```

Recommendation: Change conditional from < to <=.

Bulla: Resolved in PR 83.

LPHAC: Verified fix.

7.3.2 Creating claims and loans skips the first `ID`

Context: BullaClaimV2.sol#L150, BullaFrendLendV2.sol#L194

Description: When creating claims and by extension loans, bulla makes use of a prefix operator as can be seen below:

```
unchecked {
    claimId = ++currentClaimId;
}
```

```
_validateLoanOffer(offer, requestedByCreditor);

uint256 offerId = ++loanOfferCount;
_loanOffers[offerId] = LoanOffer({params: offer, requestedByCreditor: requestedByCreditor});
```

Prefix operators will first increment, then set the intended variable to the incremented value. Thus loan `offerId` and `claimId` will skip the 0 index. This will cause inconsistencies in the `getClaim()` and `getLoan()` as they will return valid claims for claims that didn't exist (the 0 index). There is no wider implication of this, since the creditor and debitor are both set to the default `address(0)`.

Recommendation: Change prefix operator for postfix `currentClaimId++` and `loanOfferCount++`.

Bulla: Resolved in PR 84 and PR 85.

LPHAC: Verified fixes. `currentClaimId` and `loanOfferCount` have been corrected to post-fix increment instead.

7.3.3 `getClaim` results in creditor and debtor inconsistencies after settlement

Context: BullaClaimV2.sol#L462-L476, BullaClaimV2.sol#L273C61-L273C70

Description: The creditor field value in the `Claim` struct validates the owner of the claim nft. However, upon successful payment of the claim the NFT can be transferred to the debtor as follows

```
if (claim.payerReceivesClaimOnPayment && claimPaid) _transfer(creditor, from, claimId);
```

Upon transfer completion subsequent `getClaim()` calls will return a different creditor, namely one where the `address(creditor) == address(debtor)`. This confusing state change may cause inconsistencies in the UX layers of Bulla.

Recommendation: Store and use the original creditor to avoid accounting inconsistencies.

Bulla: Resolved in PR 88.

LPHAC: Resolved fix. The claim NFT is no longer transferred to the creditor post payment, avoiding any creditor/debtor inconsistencies.

7.3.4 Loans offered via `BullaFrendLendV2` cannot be unfactored

Context: BullaFactoring.sol#L175-L272, BullaFactoring.sol#L753-L792

Description: The `BullaFactoring` contract has been modified to allow for loans to be offered by the `underwriter` account to a target `debtor` via the `offerLoan()` function. When the loan offer has been accepted, the `onLoanOfferAccepted()` function is called to store the invoice in the `activeInvoices` array.

The contract also makes `unfactorInvoice()` available to call by the original creditor of the invoice who covers any shortfall or is paid an amount when the invoice was overpaid. This same logic can't be applied to invoices created via `offerLoan()` because the original creditor is `address(this)` and therefore can't be used to cover shortfall amounts. Although, it would prove useful to have this option available with slightly different logic.

Recommendation: Consider implementing similar logic to the `unfactorInvoice()` to allow for an invoice created via `offerLoan()` to be offloaded or effectively bought by another user which would immediately return funds and owed fees up-to the current `block.timestamp`.

Bulla: I think this feature might be for another day. There is no real "originator" of the invoice other than the pool. To make a feature where the pool can sell a factored item would entail more careful and complete process to make sure it is designed as we want to be.

I think for now we will out-scope this.

LPHAC: Acknowledged that this feature might be implemented in the future.

7.3.5 `gracePeriodDays` cannot be changed while there are active invoices

Context: BullaFactoring.sol#L655-L659, BullaClaimV2.sol#L377-L409

Description: Because `gracePeriodDays` is configurable by the `onlyOwner` of the `BullaFactoring` contract, it's safe to assume that this may change in the future. There are *two* sets of loan/invoice impairment logic implemented in:

1. `BullaFrendLendV2.impairLoan()`
2. `BullaInvoice.impairInvoice()`

Both these functions effectively mark the claim's status as impaired, preventing further payments until remedied. These must be synced up with the `BullaFactoring` contract.

Recommendation: Consider adding calls via either of the above *two* functions in `BullaFactoring.impairInvoice()` depending on the invoice type and updating the `isInvoiceImpaired()` function to read `claim.impairmentGracePeriod` by adding an entry to `getInvoiceDetails()`.

Bulla: Resolved in PR 99 and PR 148.

LPHAC: Verified fixes. The `BullaFactoring` contract now uses the bulla contracts impairment grace period instead of the pool value so these values are now synced.

7.3.6 Invoice unfactoring does not reconcile all paid invoices before processing redemption queue

Context: BullaFactoring.sol#L782-L789

Description: When an invoice is approved by the underwriter and then funded by the creditor who effectively sells the claim to the `BullaFactoring` pool, the same creditor can opt to unfactor an invoice and return the NFT back to the creditor refunding any owed amount.

During this process of unfactoring, the invoice is removed from the list of active invoices and realized fees are incrementing in the contract's accounting. However, a call is made at the end of the `unfactorInvoice()` function to `_processRedemptionQueue()` without first reconciling all paid invoices. While it is true that unfactoring an invoice more or less does this for the invoice being unfactored, this is not true for other invoices.

Recommendation: Consider adding a call to `_reconcileActivePaidInvoices()` before processing the redemption queue in the `unfactorInvoice()` function.

Bulla: Resolved in PR 147.

LPHAC: Verified fix. Implemented as suggested.

7.4 Informational

7.4.1 `dueBy` parameter in `CompoundInterestLib` should be renamed

Context: CompoundInterestLib.sol

Description: The use of the `dueBy` parameter intends to signify when interest starts to accumulate on the invoice but this naming is a little confusing.

Recommendation: Consider renaming this to `interestStartDate` or similar.

Bulla: Resolved in PR 95.

LPHAC: Verified fix.

7.4.2 Redundant `numberOfPeriodsPerYear` check in `offerLoan`

Context: BullaFactoring.sol#L186

Description: There is a check in `offerLoan()` which ensures `numberOfPeriodsPerYear <= 365`, however, this is effectively already being checked via `CompoundInterestLib.validateInterestConfig()` when the loan offer is made.

Recommendation: Consider removing this redundant line.

Bulla: Resolved in PR 146.

LPHAC: Verified fix. Implemented as suggested.

7.4.3 Redundant transfer made in `_acceptLoan`

Context: BullaFrendLendV2.sol#L321-L326

Description: When is loan is accepted by either the debtor or creditor, `_acceptLoan()` performs *two* equally amounted transfers, from `offer.params.creditor -> address(this) -> finalReceiver`. This could be simplified to a single `ERC20(offer.params.token).safeTransferFrom(offer.params.creditor, finalReceiver, offer.params.loanAmount)` which would have the same effect.

Recommendation: Consider changing this to be only a single transfer or maintain as is if it is there for readability.

Bulla: Resolved in PR 91.

LPHAC: Verified fix. Implemented as suggested above.

7.5 Gas Optimization

7.5.1 Arrays are redundantly iterated through twice

Context: RedemptionQueue.sol#L179-L202, RedemptionQueue.sol#L247-L269

Description: There are *two* instances in the `RedemptionQueue` contract where an array is redundantly being iterated through twice. The first time is to count the number of instances that an array element satisfies a certain condition and the second time to populate an array of only elements which satisfy this condition. This is redundant and unnecessary and can be easily simplified.

Recommendation: Consider doing a single pass by setting the length of the array optimistically and overwriting the length value to be the target length at the end using assembly. This is efficient and doesn't leave dirty memory because the excess array elements are already zeroed.

Bulla: Resolved in PR 139.

LPHAC: Verified fix. Only a single loop is used and the array length is overwritten as per the recommendation.